

Prüfung WS 04/05 (Hausarbeit)

Christian Wilkin und Alexander Keidel

Hausarbeit

Februar 2005

Betreuung:

Prof. Dr. Peter Gemmar

Informatik

Fachbereich Design und Informatik
Fachhochschule Trier
University of Applied Sciences



FACHHOCHSCHULE TRIER

Hochschule für Technik, Wirtschaft und Gestaltung
University of Applied Sciences

Autoren: **Christian Wilkin und Alexander Keidel**
Titel: **Prüfung WS 04/05 (Hausarbeit)**
Studiengang: **Informatik**
Betreuung: **Prof. Dr. Peter Gemmar**

Februar 2005

Es wird hiermit der Fachhochschule Trier (University of Applied Sciences) die Erlaubnis erteilt, die Arbeit zu nicht-kommerziellen Zwecken zu verteilen und zu kopieren.

Unterschrift des Autors

© Copyright: Christian Wilkin und Alexander Keidel (2005)

Kurzfassung

Ein AMR soll eine Verteilungsaufgabe (RoboDeliver) selbstständig übernehmen: er soll Teile, die er an der Startpose $P_S = (0, 0, 0)$ auf einmal aufgenommen hat, an verschiedene Orte der Reihe nach verteilen.

In einer vorgegebenen Welt (siehe Abb. 0.1) sind vier Orte (Positionen $P_i = (x, y), i = \{1, \dots, 4\}$) markiert, wo die Teile abzulegen sind. Für die Verteilung der Teile erstellt der AMR eine Planung, um möglichst effizient die Teile zu verteilen (Wegstreckenoptimierung).

Unterwegs können sich dynamische Hindernisse dem AMR in den Weg stellen. Je nach Art des Hindernisses soll der AMR versuchen, das Hindernis zu umfahren oder einen neuen Pfad planen und verfolgen.

Die praktische Verteilung der Teile kann aus technischen Gründen simuliert werden: der AMR nimmt bei P_S einen Gegenstand auf, bringt ihn zur Position $P_i, i \in \{1, \dots, 4\}$, stellt ihn dort ab (um zu erkennen, dass der AMR die Position korrekt angefahren hat), nimmt ihn dann an der Ablagestelle wieder auf und fährt weiter zur Position $P_j, j \neq i \in \{1, \dots, 4\}$.

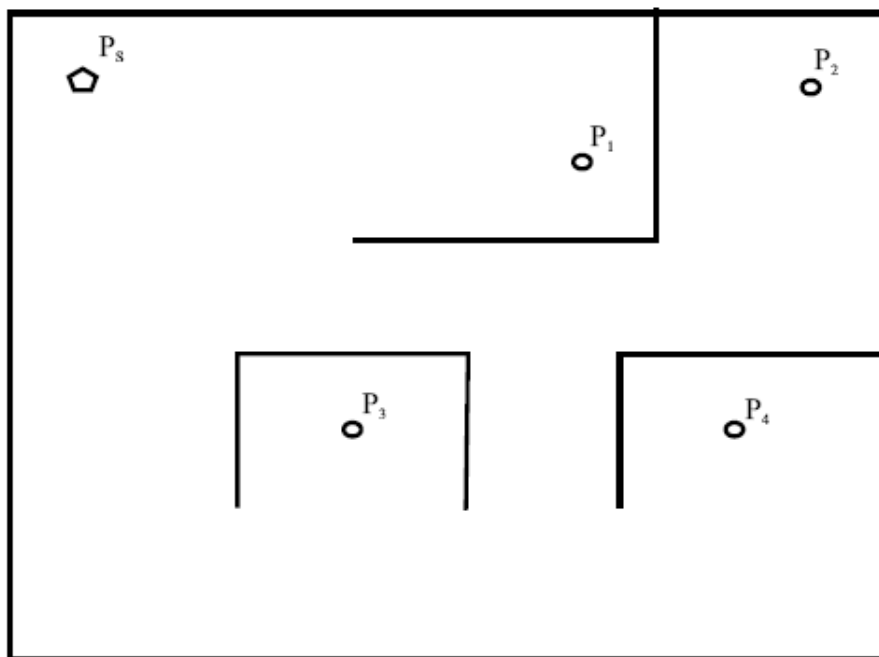


Abbildung 0.1: Weltmodell

Inhaltsverzeichnis

| | | |
|----------|--|-----------|
| 1 | Einleitung | 1 |
| 2 | Problemstellung | 2 |
| 3 | Lösungsansatz | 3 |
| 3.1 | Strategie | 3 |
| 4 | Beschreibung der verwendeten Methoden | 6 |
| 4.1 | Das Gradientenverfahren SfGrad | 6 |
| 4.1.1 | Zustände von „sfGrad“ | 7 |
| 4.1.2 | Initialisierung von „sfGrad“ | 7 |
| 4.2 | Markov-Lokalisation | 8 |
| 4.2.1 | Einsatz und Konfiguration | 8 |
| 4.3 | Der Goalmanager | 9 |
| 4.3.1 | Methode <i>addGoal()</i> | 9 |
| 4.3.2 | Methode <i>doneGoal()</i> | 9 |
| 4.3.3 | Methode <i>getGoal()</i> | 10 |
| 4.3.4 | Methode <i>changeGoal()</i> | 10 |
| 4.3.5 | Methode <i>doneAllGoals()</i> | 10 |
| 5 | Der Gripper | 11 |
| | Literaturverzeichnis | 12 |

1 Einleitung

Die hier beschriebene Hausarbeit ist im Rahmen der Vorlesung [Gem05] „Labor Robotik“ an der FH Trier entstanden.

Das „Labor Robotik“ wurde erstmals im WS02/03 durchgeführt. Es werden Problemstellungen für die Kontrolle von mobilen autonomen Robotern erarbeitet und in Form von Projekt-, Haus- und Diplomarbeiten dann praktisch umgesetzt. Die Entwicklungen werden mit der Software Saphira und dem darin enthaltenen Simulator erarbeitet und zum Abschluss mit dem Roboter Pioneer 2DX (siehe Abb.: 1.1) praktisch getestet. Diese Dokumentation wurde mit dem Satzsystem \LaTeX erstellt.



Abbildung 1.1: Roboter Pioneer 2DX

2 Problemstellung

Der Roboter hat die Aufgabe, in einer ihm bekannten Umgebung Gegenstände an bestimmten Stellen aufzunehmen und an anderen Positionen möglichst genau wieder abzulegen. Unterwegs können Hindernisse auftreten, die der Roboter umfahren muss.

Aufgabe ist es, für den Roboter Verhalten (Behaviors) zu bestimmen und zu entwickeln, die ihn in die Lage versetzen, dass er

- a) seine Transportaufgabe ökonomisch erledigt (logistische Planung)
- b) Hindernisse erkennt und geeignet umfährt (Hinderniserkennung und -vermeidung)
- c) die Gegenstände so genau anfährt, dass er sie direkt aufnehmen kann; Annahme für die Aufnahme der Gegenstände: 2D-Greifarm des Pioneer 2DX (siehe Abb.: [1.1](#))

3 Lösungsansatz

In diesem Abschnitt wird beschrieben, wie die Hauptprobleme (Navigation, Selbstlokalisierung und die Ansteuerung des Greifarms) in dieser Arbeit gelöst werden.

Das Problem der Navigation wird derart gelöst, dass dem Roboter alle notwendigen Zielpunkte vorgegeben werden, damit er seine Hauptaufgabe (Gegenstand von A nach B befördern) durchführen kann.

Die Probleme Selbstlokalisierung und Navigation sind eng miteinander verbunden. Denn nur wenn der Roboter genau weiß, wo er sich befindet, kann er die vorgegebenen Zielpunkte erreichen. In dieser Arbeit wird das Saphira-Modul für die Markov-Lokalisierung (siehe Kapitel 4.2) verwendet.

Das Problem der Ansteuerung des Greifarms wird mit Hilfe von Saphira-Funktionen gelöst. Es wird eine Schnittstelle zur Verfügung gestellt, die eine einfache Ansteuerung des Greifarms erlaubt.

Trifft der Roboter während der Fahrt auf ein dynamisches (unbekanntes) Hindernis, dann wird es umfahren. Der Roboter versucht anschließend den nächsten Zielpunkt anzufahren.

3.1 Strategie

Der AMR ¹ soll mit Hilfe des Moduls: SfGrad (siehe Kapitel 4.1) die Wege zu den Zielpositionen durch das Weltmodell planen.

Zur Realisierung der geforderten Verhaltensweisen wurde das Modell des Zustandsautomaten gewählt. In der „Hauptfire Methode“ gibt es 7 verschiedene States (Zustände).

- State 0 ist unser Anfangszustand:
In State 0 werden dem „Goalmanager“ (siehe Kapitel: 4.3) die 4 Zielpositionen angegeben. Dieser plant dann den besten Weg (eventuell auftretende dynamische Hindernisse können bei dieser Planung noch nicht in Betracht

¹kurz für „Autonome mobile Roboter“

gezogen werden), anhand der Entfernung zu den einzelnen Zielposen. Anschliessend folgt ein Übergang in Zustand „1“.

- State 1 ist der Fahrtzustand in Richtung des aktuellen Ziels:
In diesem State wird der Status des SfGrad Modules (siehe Kapitel: 4.1) abgefragt. Dieses Modul hat 5 verschiedene Zustände (siehe Kapitel: 4.1.1). Im „idle“ Zustand wird das neue Ziel mit der Funktion *sfGradSetGoal(x,y)* gesetzt. *sfGrad* wechselt somit in seinen Zustand „searching“, in welchem die Roboterpose mittels *mcUpdateRobotPose(1)* aktualisiert wird und nur noch die aktuellen (current) Sonardaten berücksichtigt werden. Wenn ein Pfad gefunden wurde, findet ein Wechsel in den Zustand „active“ statt und, der Roboter bewegt sich in Richtung des Ziels. Wenn das Ziel erreicht wurde, findet ein Übergang in den *sfGrad* Zustand „done“ statt, und es wird in unseren Zustand „2“ gewechselt. Wenn kein Pfad ermittelt werden konnte, so geht *sfGrad* in seinen Zustand „failed“ über, wo versucht wird das gleiche Ziel erneut zu ermitteln. Wenn nach n Versuchen kein Pfad zum Ziel geplant werden konnte, entscheidet der Roboter, dass das Ziel temporär nicht angefahren werden kann und teilt dieses dem „Goalmanager“ mit, der dann mit *changeGoal()* das Ziel wechselt.
- State 2 ist der Zustand indem versucht wird, den Zielpunkt möglichst genau zu erreichen, um den Gegenstand auch genau abzulegen. Als erstes wird über *mcUpdateRobotPose(1)* alle 100 ms die Pose des Roboters aktualisiert. Mit verringerter Geschwindigkeit wird der Gegenstand über die Ablegeposition navigiert. Wenn diese möglichst genau erreicht wurde, wird die Geschwindigkeit des Roboters auf „0“ gesetzt und in unseren Zustand 3 gewechselt.
- State 3 ist der Zustand indem der Gegenstand abgelegt wird, und nach einer kurzen Zeit wieder aufgenommen wird. Anschliessend wird in unseren Zustand „4“ gewechselt.
- State 4 ist der Zustand in dem sich der Roboter rückwärts um 25 cm vom Zielpunkt entfernt. Anschliessend erfolgt ein Übergang in unseren Zustand „5“.
- State 5 ist der Zustand in welchem dem „Goalmanager“ mitgeteilt wird, dass eine Zielposition erfolgreich besucht wurde. Zusätzlich wird geprüft, ob alle Ziele erreicht wurden, wenn dies der Fall ist, erfolgt ein Übergang in Zustand 6, ansonsten wird das nächste Ziel in unserem State 1 angefahren.
- State 6 ist der Zustand an dem der Roboter seine Aufgabe erledigt hat und Feierabend machen kann.

Somit ergibt sich folgender Zustandsautomat (schematische Darstellung)

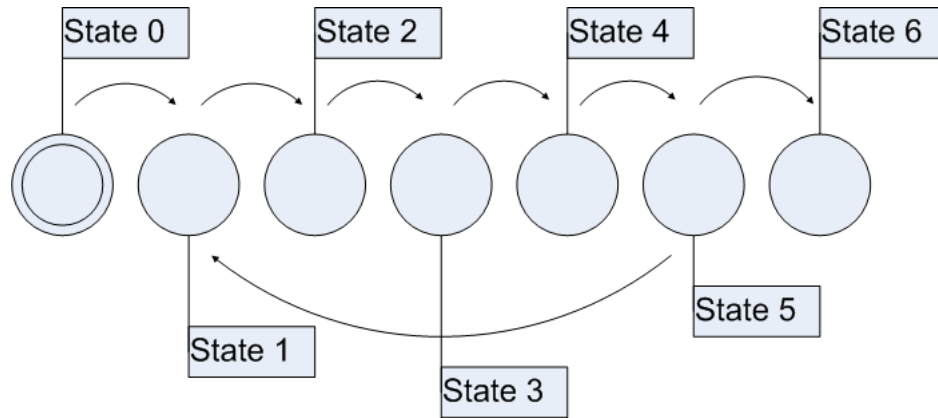


Abbildung 3.1: Zustandsautomat

4 Beschreibung der verwendeten Methoden

In diesem Abschnitt werden die Funktionsweisen der verwendeten Methoden näher beschrieben.

4.1 Das Gradientenverfahren SfGrad

Grundsätzlich ist bei der Pfadplanung zu beachten, dass zwischen befahrbaren Gebieten und unbefahrbaren Gebieten unterschieden wird. Dadurch können dynamisch auftretende Hindernisse berücksichtigt werden.

Das Gradientenverfahren in dem Modul SfGrad berücksichtigt die oben genannte Anforderung und kombiniert somit die Navigation mit der Hinderniserkennung. Sie ist auch unter dem Namen „Potentialfeldmethode“ [SiGrd05] bekannt. Der Raum wird als elektrisches Feld interpretiert, in welchem der Roboter negativ- und der Zielpunkt positiv geladen ist (siehe Abb. 4.1). Der Roboter verfolgt also durch die elektrischen Kräfte immer das Ziel (er fühlt sich zur Senke hingezogen).

Die Hindernisse sind ebenfalls negativ geladen. Dadurch kommt es zu einer Abstoßung der elektrischen Kräfte, welche sich im Umfahren bzw. Ausweichen des Hindernisses deutlich machen.

Insgesamt kann man sagen, dass der Roboter durch ein additives Feld zum Ziel geführt wird.

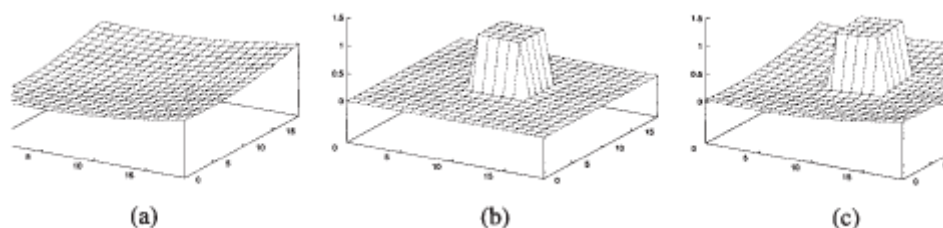


Abbildung 4.1: Potentialfelder Quelle: [SiGrd05]

- a) Potentialfeld am Ziel (Senke in der Mitte des Feldes)

- b) Hindernis Feld (Erhöhung in der Mitte des Feldes)
- c) Summe von a) und b)

4.1.1 Zustände von „sfGrad“

Das Modul `sfGrad` verfügt über 5 verschiedene Zustände [Sa03], die über die Funktion `int sfGradStatus(void)` abgefragt werden können. Die Zustände gliedern sich wie folgt auf:

- 0 steht für idle (Ruhezustand)
- 1 steht für active (Fahrzustand)
- 2 steht für done (Fahrziel erreicht)
- 3 steht für failed (Planung um zum Fahrziel zu kommen, kann nicht berechnet werden)
- 4 steht für searching (Der Pfad wird errechnet)

4.1.2 Initialisierung von „sfGrad“

Um das Modul „sfGrad“ [Sa03] nutzen zu können, muss es zuerst mit passenden Parametern initialisiert werden. Dies geschieht über die folgenden Angaben:

- `sfGradInitRes(a,b)` → a ist Zellenabstand in mm, b ist der Radius der gewählt wird, wenn versucht wird die Zielpose genau zu erreichen.
- `sfGradSetMap(mcGetObject())` → liefert die Karte zurück, die vom Lokalisationsmodul benutzt wird.
- `sfGradSetSonarBuffer(1)` → Welche Readings sollen ausgewertet werden (1 = current, 2 = kommutierte, 3 = beide).
- `sfGradUseArtifacts(true)` → Sollen die Artefakte aus der *.wld Datei benutzt werden?
- `sfGradSetSpeed(a,b,c)` - Setzt die Geschwindigkeiten (a = maximale Geschwindigkeit, b = mittlere Geschwindigkeit [wenn Objekte in der Nähe], c = Rückwärts Geschwindigkeit).
- `sfGradSetMax(x,y)` → Gibt die maximale Grösse des Gradienten-Fensters an.

- $sfGradsetDone(a,b) \rightarrow a =$ Entfernung zum Zielpunkt, ab welcher die Geschwindigkeit auf mittlere Geschwindigkeit gesetzt wird, $b =$ Entfernung, wenn nah genug am Zielpunkt.
- $sfGradObsParams(x,y) \rightarrow x =$ Entfernung, die auf keinen Fall zu einem Objekt unterschritten werden darf, $y =$ Entfernung, ab der versucht wird ein Hindernis zu umfahren.

4.2 Markov-Lokalisation

Die sogenannte Markov-Lokalisation [SiMar05] ermöglicht eine globale Selbstlokalisierung. Notwendig dazu ist eine Karte und die Ultraschallsensoren des AMRs. Die Distanz zu den Hindernissen wird ständig mit der Karte verglichen. Durch eine Diskretisierung der Bewegung des AMR in einzelne Schritte, kann eine höhere Genauigkeit der Positionsschätzung erreicht werden.

Die Markov-Lokalisation berechnet eine 2 Dimensionale Hypothesenwolke, deren Dichte ein Maß für die Aufenthaltswahrscheinlichkeit ist. Diese ist notwendig, damit mögliche Ambiguitäten vermieden werden können. Wenn die Position mit hoher Wahrscheinlichkeit bekannt ist, so wird diese ständig verfolgt.

Bei Abbruch wird eine Relokalisierung durchgeführt.

4.2.1 Einsatz und Konfiguration

Das Modul ist in zwei DLLs aufgeteilt. `sfLoc.dll` und `sfLocFl.dll` befinden sich beide im „lib“ Verzeichnis von Saphira. In der ersten ist der Kern der Lokalisierung implementiert, während letztere die GUI von Saphira um ein „Localize“ Menü erweitert. Am besten lädt man beide DLLs und die Karte (World File) zusammen in der Initialisierungsdatei „start.act“.

Listing 1: Einbinden in das Act-File

```
1 loadworld hausarbeit ;
2 loadlib sfLoc ;
3 loadlib sfLocFl ;
4 mcSonarInit () ;
```

Der letzte Befehl sorgt dafür, dass im Update-Schritt Sonardaten benutzt werden. Außerdem wird das Worldfile analysiert und eine ML Karte daraus erstellt. Es können einige Parameter der Lokalisierung sowohl über das Menü Localize -> Parameters als auch über C++/Colbert Funktionen verändert werden.

4.3 Der Goalmanager

Der Goalmanager verwaltet die abzufahrenden und die bereits erreichten Positionen in 2 Listen.

- Bei der Initialisierung wird die Liste der abzufahrenden Positionen mit allen Zielpositionen gefüllt (Methode *addGoal()*).
- Wurde ein Gegenstand abgelegt, wird dies mit der Methode *doneGoal()* dem Goalmanager mitgeteilt, welcher dann die Listen entsprechend verschiebt.
- Falls sfGrad keinen Pfad zu einem Zielpunkt finden kann, wird die Methode *changeGoal()* aufgerufen.
- Die nächste Zielpose wird über die Methode *getGoal()* als ArPose zurückgegeben.
- Die Methode *doneAllGoals()* liefert *true* zurück, falls die Liste „goals“ leer ist, und somit alle Ziele angefahren wurden.

4.3.1 Methode *addGoal()*

In dieser Methode wird eine Pose an die Liste „goals“ angehängen.

Listing 2: Die Methode *addGoal()*

```
1 void akcwGoalManager::addGoal(ArPose pose)
2 {
3     goals.push_back(pose);
4 }
```

4.3.2 Methode *doneGoal()*

In dieser Methode wird die aktuelle Pose in die Liste „goalsDone“ eingefügt, und wenn die Liste „goals“ nicht leer ist, wird die aktuelle Pose von vorne entfernt.

Listing 3: Die Methode *doneGoal()*

```
1 void akcwGoalManager::doneGoal(ArPose pose)
2 {
3     goalsDone.push_front(*goals.begin());
4     if (goals.size() != 0)
5         goals.pop_front();
6 }
```

4.3.3 Methode *getGoal()*

In dieser Methode wird die erste ArPose aus der Liste „goals“ zurückgegeben.

Listing 4: Die Methode *getGoal()*

```
1 ArPose akcwGoalManager: getGoal()
2 {
3     return goals.front();
4 }
```

4.3.4 Methode *changeGoal()*

In dieser Methode wird die erste ArPose aus der Liste „goals“ entnommen und hinten wieder angehängt.

Listing 5: Die Methode *changeGoal()*

```
1 void akcwGoalManager: changeGoal()
2 {
3     ArPose pose;
4     pose = goals.front();
5     goals.pop_front();
6     goals.push_back(pose);
7 }
```

4.3.5 Methode *doneAllGoals()*

In dieser Methode wird eine Pose an die Liste „goals“ angehängt.

Listing 6: Die Methode *doneAllGoals()*

```
1 bool akcwGoalManager:: doneAllGoals()
2 {
3     if (goals.size() == 0)
4         return true;
5     else
6         return false;
7 }
```

5 Der Gripper

Hat der mobile Roboter seine Route geplant, seine Position bestimmt und den Pfad zu einem Ziel gefunden, kann ein Ziel angefahren werden. Die Greiferein-

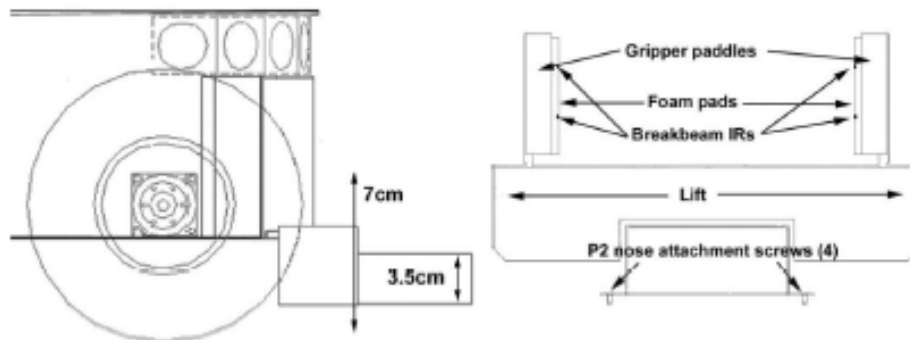


Abbildung 5.1: Greifereinheit des 2DX

heit des P2DX, gekennzeichnet durch zwei zusammenschiebbare Riegel/Schaukeln („paddles“) und einen Lift. Die Riegel reagieren auf Druck und „merken“ so, ob das Objekt gegriffen wurde, oder ob es noch nicht fest gehalten wird. Im State 3 kommt der Gripper zum Einsatz: Lift senken, Riegel öffnen, Lift anheben und um 30cm zurücksetzen, um das Objekt nicht umzuwerfen.

Literaturverzeichnis

- [Gem05] Gemmar, Peter (2004): *Vorlesung Robotik WS 04/05*, Vorlesungsskript, FH-Trier, Seite 22-27
- [Sa03] Dimitri van Heesch (2003): *Saphira Dokumentation*, Gradient Navigation Module, Active Media, [Saphira Doku](#)
- [SiMar05] Simon, Dirk (2005): *TippsTricksRobolab*, Die Markov Lokalisation, FH-Trier, Seite 11-12
- [SiGrd05] Simon, Dirk (2005): *TippsTricksRobolab*, Das Gradientenverfahren SfGrad, FH-Trier, Seite 12-14